**Caption Motion and the Observer Patterns**

The caption in our GUI is a JLabel that listens for text changes from the JTextField labeled “Caption:”. Since we display the text using an object that extends JComponent, we utilize the Observer pattern by attaching listeners to it.

There are two objects that must listen for caption motion. The first is obviously the caption itself; the second more subtly is the object that stores the caption location to the slideshow file. Both will need to be in sync with each other; to achieve this, we created a nested, abstract class “CaptionLabelMouseInputAdapter” that extends the Java abstract class MouseInputAdapter (see or the file “SlideShowImagePanel.java” for the source code). This abstract class encapsulates all necessary mouse motion calculations into a single location, which the two objects that must listen for mouse motion then extend to include the additional functionality they individually need.

The caption extends CaptionLabelMouseInputAdapter through an anonymous class in function “addMouseInputListenerToCaption” (see or the file “SlideShowImagePanel.java” for the source code). The only interface method the caption JLabel needs to be extend is “mouseDragged”; in this function, if the caption has moved (as calculated by the abstract class functions), the caption location is updated/repainted. After the anonymous class object is created, it is added to the caption JLabel through functions “addMouseListener” and “addMotionMouseListener”.

The file contents tracker similarly extends CaptionLabelMouseInputAdapter through an anonymous class in function “addMouseInputListenerToCaptionForFileContents”, which is in the file “SlideShowContentsPanel.java” and included as a courtesy in Appendix C. The only interface method that needed to be extended was “mouseReleased” since the slideshow file contents are only concerned with the final location of the caption after dragging and not any intermediary locations. The function, addMouseInputListenerToCaptionForFileContents, is called in the GUI’s main function after creating the JPanel, which contains the displayed image and caption.

**Summary:** All mouse inputs and tracking is done using listeners; this approach utilizes the Observer pattern.

**The Undo Buffer and the Command and Observer Patterns**

The Command pattern encapsulates actions (e.g. user interaction) into an object. In alignment with this approach, we created an interface type “SlideShowGUICommand”, which will serve as the supertype of any undo-able actions (see or file “SlideShowGUICommand.java” for the source code); this interface has two methods “execute” and “undo”. By using an interface type, we could have different undoable operations handled using a single architecture.

Since the undo stack has a maximum depth (e.g. 10) and the oldest object (i.e. bottom of the stack) is deleted whenever a new object is added to a full stack, we implemented our undo stack using a circular buffer; at its lowest level, the buffer is an ArrayList that stores objects of the interface type “SlideShowGUICommand”. This functionality is encapsulated with the Undo JMenuItem in the SlideShowEditJMenu class.

Saved changes to a slide is the only supported undoable command. To encapsulate this functionality, we created the SaveGUICommand class, which implements the SlideShowGUICommand interface (see or file “SlideShowContentsPanel.java” for the source code). In order to undo saved changes to a slide, the SlideShowGUICommand object stores the previous slide information (see variable previousImageInstance). To restore the previous image information, an object of type SlideShowGUICommand only needs to call the undo method.

Changes are saved to a slide when a user does one of two events. The first is when an actor presses the “Save Image” button; the Save Image button has an ActionListener attached to it (see function “actionPerformed” in or the file “SlideShowContentsPanel.java” for the source code). The function only updates the Undo buffer if the slide contents (e.g. caption location, caption text, image path, etc.) have changed. If the slide has changed, it creates a SlideShowGUICommand object, adds that object to the Undo buffer, and then invokes the execute method on the object to update the slide contents. The second type of event that causes a slide to be saved is when a slide caption is moved. When that occurs, only the slide’s caption location is updated (i.e. any image path or caption text changes are not saved). This is handled by an anonymous object that extends the CationLabelMouseInputAdapter abstract class (see function “addMouseInputListenerToCaptionForFileContents” in Appendix C or the file “SlideShowContentsPanel.java”).

The “Undo” menu selection is a JMenuItem that is part of the overall “Edit” menu. We created a custom class “SlideShowEditJMenu” to implement the undo functionality (see or the source code in file “SlideShowEditJMenu.java”). There is an ActionListener attached to the Undo JMenuItem; it is created in the method “createUndoActionListener” (see or file for the source code). Whenever the Undo button is pressed, it removes one SlideShowGUICommand object from the Undo buffer (see class UndoBuffer in or in “SlideShowEditJMenu.java”), and then calls the undo method on that object. The ActionListener then checks if the undo buffer is empty; if it is, the listener disables the “Undo” button.

The Undo menu also has a listener attached to the “New” menu option. The listener is created in the method “generateClearUndoBufferActionListener” (see or the file “SlideShowEditJMenu.java” for the source code). Similarly, the Undo menu has ActionListeners attached to the JFileChooser objects for the “Open” and “Save” JMenuItems; these ActionListener objects are created in the method “generateFileChooserClearUndoBufferActionListener” (see or the file “SlideShowEditJMenu.java” for the source code). The three previously described ActionListeners are added to the JMenuItem objects in the constructor for the SlideShowJMenuBar (see function “addActionListener” in or “SlideShowJMenuBar.java”).

**Summary:** By encapsulating slide save operations and in turn undo operations into objects, we followed the Command pattern. Moreover, all GUI inputs associated with the undo functionality are handled using listeners, as dictated by the observer pattern.

Appendix – CaptionLabelMouseInputAdapter Source Code

**public** **abstract** **class** CaptionLabelMouseInputAdapter **extends** MouseInputAdapter{

//---- Store initial information about the caption

**private** **int** initialCaptionX;

**private** **int** initialCaptionY;

//---- Store the last mouse position

**private** **int** lastMouseX;

**private** **int** lastMouseY;

//---- Store Caption information

**private** **int** latestCaptionX;

**private** **int** latestCaptionY;

**private** **boolean** captionMoved;

//----

**boolean** mouseOutsideValidArea;

/\*\*

\* Stores the initial X and Y location when the mouse is pressed.

\*/

@Override

**public** **void** mousePressed(MouseEvent e){

//---- Initially latest and initial position are the same as the the caption's current position

latestCaptionX = initialCaptionX = captionLabel.getX();

latestCaptionY = initialCaptionY = captionLabel.getY();

//----- Get the mouse location information

lastMouseX = e.getXOnScreen();

lastMouseY = e.getYOnScreen();

//---- By default caption not moved

captionMoved = **false**;

//---- Mouse still in valid area

mouseOutsideValidArea = **false**;

}

@Override

**public** **void** mouseDragged(MouseEvent e){

//---- Check if the mouse position is invalid

**if**(mouseOutsideValidArea && getMousePosition(**true**) == **null**) **return**;

//---- Get the caption location

Point captionLocation = captionLabel.getLocation();

**int** captionXLoc = (**int**)captionLocation.getX();

**int** captionYLoc = (**int**)captionLocation.getY();

//---- Get the newX and newY locations for the caption label

**int** newX = captionXLoc + (e.getXOnScreen() - lastMouseX);

**int** newY = captionYLoc + (e.getYOnScreen() - lastMouseY);

//---- Handle the case where the cursor just reentered the valid space

**if**(mouseOutsideValidArea){

//--- Handle default case where the mouse did not re-enter the valid space.

**if**(newX < captionLabel.getMinimumXLocation() || newX > captionLabel.getMaximumXLocation())

newX = captionXLoc + (e.getX());

**if**(newY < captionLabel.getMinimumYLocation() || newY > captionLabel.getMaximumYLocation())

newY = captionYLoc + (e.getY());

}

//----- Ensure the mouse is not too far away from the caption

Point mousePositionInPanel = getMousePosition(**true**);

**if**(mousePositionInPanel != **null**){

//----- Ensure the mouse and CaptionLabel are not separated too much in the X direction.

**int** mouseComponentXDistance = (**int**)mousePositionInPanel.getX() - newX;

**if**( mouseComponentXDistance < 0 || mouseComponentXDistance > captionLabel.getWidth()){

newX += mouseComponentXDistance;

}

//----- Ensure the mouse and CaptionLabel are not separated too much in the Y direction.

**int** mouseComponentYDistance = (**int**)mousePositionInPanel.getY() - newY;

**if**( mouseComponentYDistance < 0 || mouseComponentYDistance > captionLabel.getHeight()){

newY += mouseComponentYDistance;

}

}

//---- Update X location

**if**(newX < captionLabel.getMinimumXLocation()){

newX = captionLabel.getMinimumXLocation();

}

**else** **if**(newX > captionLabel.getMaximumXLocation()){

newX = captionLabel.getMaximumXLocation();

}

lastMouseX = e.getXOnScreen();

latestCaptionX = newX;

//---- Update Y location

**if**(newY < captionLabel.getMinimumYLocation()){

newY = captionLabel.getMinimumYLocation();

}

**else** **if**(newY > captionLabel.getMaximumYLocation()){

newY = captionLabel.getMaximumYLocation();

}

lastMouseY = e.getYOnScreen();

latestCaptionY = newY;

//----- Check if the caption moved. May not move if you are at the boundary.

**if**(captionXLoc != newX || captionYLoc != newY) captionMoved = **true**; //---- Mark caption moved.

//---- Check if the mouse left the valid area

**if**(getMousePosition(**true**) == **null**)

mouseOutsideValidArea = **true**;

**else**{

mouseOutsideValidArea = **false**;

}

}

/\*\*

\* Gets the initial X and Y location of the Caption.

\* **@return** Point containing initial X and Y location of the captionLabel

\*/

**public** Point getInitialCaptionLocation(){ **return** **new** Point(initialCaptionX, initialCaptionY); }

/\*\*

\* Gets the final X and Y location of the Caption.

\* **@return** Point containing final X and Y location of the captionLabel

\*/

**public** Point getFinalCaptionLocation(){ **return** **new** Point(latestCaptionX, latestCaptionY); }

/\*\*

\* After the mouse has been released, this function is used to determine whether the caption moved.

\* **@return** Boolean value of whether the caption moved during the mouse dragging

\*/

**public** **boolean** getDidCaptionMove(){ **return** captionMoved; }

}

Appendix – addMouseInputListenerToCaption Source Code

/\*\*

\* Creates a MouseListener and MouseMotionListener and adds it to CaptionLabel.

\*/

**private** **void** addMouseInputListenerToCaption(){

//-- Create an anonymous object to listen for mouse motions.

MouseInputAdapter captionListener = **new** CaptionLabelMouseInputAdapter(){

@Override

**public** **void** mouseDragged(MouseEvent e){

//---- Calculate the new caption locations

**super**.mouseDragged(e);

//----- If the caption moved, update its location

**if**(getDidCaptionMove()){

captionLabel.setLocation(**this**.getFinalCaptionLocation());

}

}

};

//----- Create the mouse listener

captionLabel.addMouseListener(captionListener);

captionLabel.addMouseMotionListener(captionListener);

}

Appendix – addMouseInputListenerToCaptionForFileContents Source Code

/\*\*

\* Creates a mouse motion listener to extract the caption location for saving to the file.

\*/

**public** **void** addMouseInputListenerToCaptionForFileContents(SlideShowImagePanel imagePanel){

//-- Create an anonymous object to listen for mouse motions.

MouseInputAdapter captionListener = imagePanel.**new** CaptionLabelMouseInputAdapter(){

@Override

**public** **void** mouseReleased(MouseEvent e){

**super**.mouseReleased(e);

**int** selectedIndex = *slideShowList*.getSelectedIndex();

//---- Check if any image was selected and whether the caption moved

**if**(selectedIndex > -1 && **this**.getDidCaptionMove()){

captionLocation = **this**.getFinalCaptionLocation();

//----- Store the previous image instance

SlideShowImageInstance previousImageInstance = *slideShowFileContents*.getImageInstance(selectedIndex);

SlideShowImageInstance newImageInstance = **new** SlideShowImageInstance( selectedIndex+1,

previousImageInstance.getImagePath(),

previousImageInstance.getImageCaption(),

captionLocation);

//---- If no change in the image instance, do nothing. Do not add to undo queue.

**if**(previousImageInstance.equals(newImageInstance)) **return**;

//----- Create a command for executing and undoing the save command.

SlideShowGUICommand newSaveImageCommand = **new** SaveGUICommand(newImageInstance, previousImageInstance);

newSaveImageCommand.execute();

((SlideShowJMenuBar)undoBuffer).addCommandToUndoBuffer(newSaveImageCommand);

}

}

};

imagePanel.addCaptionMouseInputListener(captionListener);

}

Appendix – SlideShowGUICommand Source Code

/\*\*

\* Interface type used to encapsulate undo-able actions.

\*

\* **@author** Zayd

\*

\*/

**public** **interface** SlideShowGUICommand {

/\*\*

\* Needs to be extended to perform the action.

\*/

**public** **void** execute();

/\*\*

\* Needs to be extended to undo the action performed by the "execute" method.

\*/

**public** **void** undo();

}

Appendix – SaveGUICommand Source Code

/\*\*

\*

\* This class is used to Save and Unsave Image Instance Saves.

\*

\* **@author** Zayd

\*

\*/

**private** **class** SaveGUICommand **implements** SlideShowGUICommand{

**private** SlideShowImageInstance previousImageInstance;

**private** SlideShowImageInstance newImageInstance;

/\*\*

\*

\* **@param** newImageInstance New Image Image Instance to be set.

\* **@param** previousImageInstance Previous Image Image Instance to be set.

\*/

**public** SaveGUICommand(SlideShowImageInstance newImageInstance,

SlideShowImageInstance previousImageInstance){

//---- Copy make clones of the input parameters.

**this**.newImageInstance = (SlideShowImageInstance)(newImageInstance.clone());

**this**.previousImageInstance = (SlideShowImageInstance)(previousImageInstance.clone());

}

@Override

**public** **void** execute() {

*slideShowFileContents*.setImageInstance(newImageInstance);

*slideShowListModel*.setElementAt(*slideShowFileContents*.getImageInstance(newImageInstance.getImageID()-1),

newImageInstance.getImageID()-1);

}

@Override

**public** **void** undo() {

*slideShowFileContents*.setImageInstance(previousImageInstance);

*slideShowListModel*.setElementAt(*slideShowFileContents*.getImageInstance(previousImageInstance.getImageID()-1),

previousImageInstance.getImageID()-1);

//---- Deselect then reselect the list item to get GUI to refresh

**if**(previousImageInstance.getImageID() - 1 == *slideShowList*.getSelectedIndex()){

*slideShowList*.clearSelection();

*slideShowList*.setSelectedIndex(previousImageInstance.getImageID() - 1);

}

}

}

Appendix – SlideShowContentsPanel actionPerformed Source Code

/\*\*

\* Internal function to handle all action listeners (e.g. Add New button, Save Image button, etc.) for this panel.

\*/

**public** **void** actionPerformed(ActionEvent e){

**if**(e.getActionCommand().equals(*ADD\_NEW\_BUTTON\_COMMAND\_NAME*)){

//---- Create a new image

*slideShowFileContents*.addNewImageInstance();

**int** newImageIndex = *slideShowFileContents*.getNumberOfImageInstances()-1;

//--- Add the element and select it and make sure it is visible.

*slideShowListModel*.addElement(*slideShowFileContents*.getImageInstance(newImageIndex));

*slideShowList*.setSelectedIndex(newImageIndex); //---- Uses base 0 so subtract one

*slideShowList*.ensureIndexIsVisible(newImageIndex);

//---- Set a default caption location.

captionLocation = **new** Point(SlideShowImageInstance.*DEFAULT\_IMAGE\_LOCATION*,

SlideShowImageInstance.*DEFAULT\_IMAGE\_LOCATION*);

}

**else** **if**(e.getActionCommand().equals(*SAVE\_BUTTON\_COMMAND\_NAME*)){

**int** selectedIndex = *slideShowList*.getSelectedIndex();

//---- Check if any image was selected.

**if**(selectedIndex == -1){

JOptionPane.*showMessageDialog*(**null**, "Before a changes can be saved to a slideshow, a specific image must be selected \n"

+ "from the list of images. Please select an image from the list, and try again.");

**return**;

}

//----- Store the previous image instance

SlideShowImageInstance previousImageInstance = *slideShowFileContents*.getImageInstance(selectedIndex);

SlideShowImageInstance newImageInstance = **new** SlideShowImageInstance( selectedIndex+1, *fileBrowserText*, *captionText*,

captionLocation);

//---- If no change in the image instance, do nothing. Do not add to undo queue.

**if**(previousImageInstance.equals(newImageInstance)) **return**;

//----- Create a command for executing and undoing the save command.

SlideShowGUICommand newSaveImageCommand = **new** SaveGUICommand(newImageInstance, previousImageInstance);

newSaveImageCommand.execute();

((SlideShowJMenuBar)undoBuffer).addCommandToUndoBuffer(newSaveImageCommand);

// //----- Update the image instance information

// slideShowFileContents.setImageInstance(selectedIndex, fileBrowserText, captionText,

// (int)captionLocation.getX(), (int)captionLocation.getY());

// slideShowListModel.setElementAt(slideShowFileContents.getImageInstance(selectedIndex), selectedIndex);

}

}

Appendix – SlideShowEditJMenu Source Code

**public** **class** SlideShowEditJMenu **extends** JMenu {

/\*\*

\*

\*/

**private** **static** **final** **long** *serialVersionUID* = -871826944474861418L;

**public** **static** **final** **int** *UNDO\_DEPTH* = 10;

**public** **static** JMenuItem *undoMenuItem*;

**public** **static** UndoBuffer<SlideShowGUICommand> *undoBuffer*;

**public** SlideShowEditJMenu(String menuName, String undoMenuItemName){

//---- Create the edit menu

**super**(menuName);

//----- Add the JMenuItem

*undoMenuItem* = **new** JMenuItem(undoMenuItemName);

*undoMenuItem*.setEnabled(**false**); //---- By default enable is disabled

*undoMenuItem*.setActionCommand("editMenu" + undoMenuItemName);

createUndoActionListener();//---- Creates an ActionListener for when the "Undo" menu item is pressed.

**this**.add(*undoMenuItem*); //---- Add the menu item to the list

//---- Initialize the undo buffer

*undoBuffer* = **new** UndoBuffer<SlideShowGUICommand>(*UNDO\_DEPTH*);

}

/\*\*

\* Creates an ActionListener for the Undo Menu Item and then adds it to the menu item.

\*/

**private** **void** createUndoActionListener(){

ActionListener undoActionListener = **new** ActionListener(){

/\*\*

\* Gets the last action performed and then undoes it.

\*/

@Override

**public** **void** actionPerformed(ActionEvent e){

SlideShowGUICommand lastCommand = *undoBuffer*.remove();

lastCommand.undo();

//---- If the buffer is empty, disable the menu item.

**if**(*undoBuffer*.isEmpty()){

*undoMenuItem*.setEnabled(**false**);

}

}

};

//--- Add the ActionListener

*undoMenuItem*.addActionListener(undoActionListener);

}

/\*\*

\* Adds another item to the undo buffer.

\*

\* **@param** newCommand New command to be added to the UndoBuffer

\*/

**public** **void** addCommandToUndo(SlideShowGUICommand newCommand){

*undoBuffer*.add(newCommand);

//---- Any time you add an item to the menu, it is guaranteed not be be empty.

*undoMenuItem*.setEnabled(**true**);

}

/\*\*

\* Generates and returns an action listener that can be used

\* **@return** New action listener that can be used to clear the Undo Buffer.

\*/

**public** ActionListener generateClearUndoBufferActionListener(){

**return** **new** ActionListener(){

@Override

**public** **void** actionPerformed(ActionEvent e) {

clearUndoBuffer();

}

};

}

/\*\*

\*

\* **@return** New action listener that is specifically intended for JFileChoosers and can clear the UndoBuffer.

\*/

**public** ActionListener generateFileChooserClearUndoBufferActionListener() {

**return** **new** ActionListener(){

@Override

**public** **void** actionPerformed(ActionEvent e) {

//---- Do not do anything on an cancelled command

**if**(e.getSource() **instanceof** JFileChooser && e.getActionCommand().equals(JFileChooser.*CANCEL\_SELECTION*) ){

**return**;

}

clearUndoBuffer();

}

};

}

**public** **void** clearUndoBuffer(){

*undoBuffer*.reset();

//---- Disable the Undo Menu Item.

*undoMenuItem*.setEnabled(**false**); //---- By default enable is disabled

}

**private** **class** UndoBuffer<E>{

ArrayList<E> bufferItems;

**int** bufferItemCount;

**int** bufferHead;

**int** bufferTail;

**public** UndoBuffer(**int** size){

//--- Create an array list of the specified size

bufferItems = **new** ArrayList<E>(size);

//---- Initialize the buffer as if it was a clear. Leveraging other source code.

reset();

//---- Populate the array with null

**for**(**int** i = 0; i < size; i++){

bufferItems.add(**null**);

}

}

/\*\*

\* Adds an item to the buffer. If the buffer is full, it overwrites the oldest item in the queue.

\*

\* **@param** item Adds this item to the buffer

\*/

**public** **void** add(E item){

//---- Update the buffer head

bufferHead = (++bufferHead) % bufferItems.size();

//---- Get the item

bufferItems.set(bufferHead, item);

//---- Decide if queue is full, move the tail. Otherwise increment the item count

**if**(bufferItemCount == bufferItems.size()){

bufferTail = (++bufferTail) % bufferItems.size();

}

**else**{

bufferItemCount++;

}

}

**public** E remove(){

//---- If the buffer is empty, return null.

**if**(isEmpty()){

**return** **null**;

}

//---- Gets the item that will be returned.

E itemtoReturn = bufferItems.get(bufferHead);

//---- If necessary roll back across the head of the

bufferHead = ((--bufferHead) >= 0 ) ? bufferHead : bufferItems.size() - 1 ;

//---- Decrement the number of items in the buffer

bufferItemCount--;

//---- return the value

**return** itemtoReturn;

}

/\*\*

\* Gets whether the buffer has any elements in it.

\*

\* **@return** True if buffer is empty, false otherwise

\*/

**public** **boolean** isEmpty(){

**if**(bufferItemCount==0) **return** **true**;

**return** **false**;

}

**public** **void** reset(){

//bufferItems.clear();

//---- Initialize the locations on the buffer

bufferHead = -1;

bufferTail = 0;

bufferItemCount = 0;

}

}

}

Appendix – createUndoActionListener Source Code

/\*\*

\* Creates an ActionListener for the Undo Menu Item and then adds it to the menu item.

\*/

**private** **void** createUndoActionListener(){

ActionListener undoActionListener = **new** ActionListener(){

/\*\*

\* Gets the last action performed and then undoes it.

\*/

@Override

**public** **void** actionPerformed(ActionEvent e){

SlideShowGUICommand lastCommand = *undoBuffer*.remove();

lastCommand.undo();

//---- If the buffer is empty, disable the menu item.

**if**(*undoBuffer*.isEmpty()){

*undoMenuItem*.setEnabled(**false**);

}

}

};

//--- Add the ActionListener

*undoMenuItem*.addActionListener(undoActionListener);

}

Appendix – UndoBuffer Class Source Code

**private** **class** UndoBuffer<E>{

ArrayList<E> bufferItems;

**int** bufferItemCount;

**int** bufferHead;

**int** bufferTail;

**public** UndoBuffer(**int** size){

//--- Create an array list of the specified size

bufferItems = **new** ArrayList<E>(size);

//---- Initialize the buffer as if it was a clear. Leveraging other source code.

reset();

//---- Populate the array with null

**for**(**int** i = 0; i < size; i++){

bufferItems.add(**null**);

}

}

/\*\*

\* Adds an item to the buffer. If the buffer is full, it overwrites the oldest item in the queue.

\*

\* **@param** item Adds this item to the buffer

\*/

**public** **void** add(E item){

//---- Update the buffer head

bufferHead = (++bufferHead) % bufferItems.size();

//---- Get the item

bufferItems.set(bufferHead, item);

//---- Decide if queue is full, move the tail. Otherwise increment the item count

**if**(bufferItemCount == bufferItems.size()){

bufferTail = (++bufferTail) % bufferItems.size();

}

**else**{

bufferItemCount++;

}

}

**public** E remove(){

//---- If the buffer is empty, return null.

**if**(isEmpty()){

**return** **null**;

}

//---- Gets the item that will be returned.

E itemtoReturn = bufferItems.get(bufferHead);

//---- If necessary roll back across the head of the

bufferHead = ((--bufferHead) >= 0 ) ? bufferHead : bufferItems.size() - 1 ;

//---- Decrement the number of items in the buffer

bufferItemCount--;

//---- return the value

**return** itemtoReturn;

}

/\*\*

\* Gets whether the buffer has any elements in it.

\*

\* **@return** True if buffer is empty, false otherwise

\*/

**public** **boolean** isEmpty(){

**if**(bufferItemCount==0) **return** **true**;

**return** **false**;

}

**public** **void** reset(){

//bufferItems.clear();

//---- Initialize the locations on the buffer

bufferHead = -1;

bufferTail = 0;

bufferItemCount = 0;

}

}

Appendix – generateClearUndoBufferActionListener Source Code

/\*\*

\* Generates and returns an action listener that can be used

\* **@return** New action listener that can be used to clear the Undo Buffer.

\*/

**public** ActionListener generateClearUndoBufferActionListener(){

**return** **new** ActionListener(){

@Override

**public** **void** actionPerformed(ActionEvent e) {

clearUndoBuffer();

}

};

}

Appendix – generateFileChooserClearUndoBufferActionListener Source Code

/\*\*

\*

\* **@return** New action listener that is specifically intended for JFileChoosers and can clear the UndoBuffer.

\*/

**public** ActionListener generateFileChooserClearUndoBufferActionListener() {

**return** **new** ActionListener(){

@Override

**public** **void** actionPerformed(ActionEvent e) {

//---- Do not do anything on an cancelled command

**if**(e.getSource() **instanceof** JFileChooser && e.getActionCommand().equals(JFileChooser.*CANCEL\_SELECTION*) ){

**return**;

}

clearUndoBuffer();

}

};

}

Appendix – File JMenuItem addActionListener Function Source Code

/\*\*

\*

\* Allows for ActionListeners to be added to the Menu Bar. Menu Items that support listeners are "New", "Open", and "Save".

\*

\* **@param** listener ActionListener to be added.

\* **@param** objectType Type of Listener to be added.

\*/

**public** **void** addActionListener( ActionListener listener, **int** objectType ){

**switch**(objectType){

**case** *NEW\_FILE\_LISTENER*:

//---- Iterate through all the menu items

**for**(Component menuComponent : *fileMenu*.getMenuComponents()){

JMenuItem menuItem = (JMenuItem)menuComponent;

//---- Check if the "New" Menu Item

**if**(objectType == *NEW\_FILE\_LISTENER* && menuItem.getText().toLowerCase().equals("new")){

menuItem.addActionListener(listener);

**return**;

}

}

**return**;

**case** *OPEN\_FILE\_LISTENER*:

*openFileChooser*.addActionListener(listener);

**return**;

**case** *SAVE\_FILE\_LISTENER*:

*saveFileChooser*.addActionListener(listener);

**return**;

}

}